VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE CHOVÁNÍ SÍTĚ NA ZÁKLADĚ ANALÝZY KONFIGURAČNÍCH SOUBORŮ AKTIVNÍCH SÍŤOVÝCH ZAŘÍZENÍ

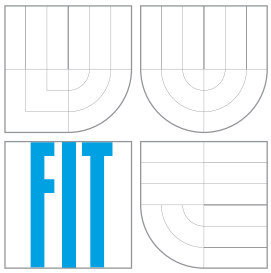
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER SCHERFEL

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE CHOVÁNÍ SÍTĚ NA ZÁKLADĚ ANALÝZY KONFIGURAČNÍCH SOUBORŮ AKTIVNÍCH SÍŤOVÝCH ZAŘÍZENÍ

SIMULATION OF NETWORK BEHAVIOUR BASED ON ANALYSIS OF CONFIGURATION OF ACTIVE NETWORK DEVICES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETER SCHERFEL
AUTHOR

VEDOUCÍ PRÁCE — Ing. ONDŘEJ RYŠAVÝ, Ph.D.
SUPERVISOR

BRNO 2009

Abstrakt

V této bakalářské práci se zabýváme simulováním počítačových sítí v simulačním nástroji OMNeT++. Vytváříme nástroj, který na základe analýzy konfiguračních souborů vytvoří všechna potřebna nastavení pro simulaci. Je zde popsána gramatika konfigurčního souboru Cisco směrovačů. Tato práce také obsahuje informace a návod na rozšíření stávající gramatiky o nové příkazy.

Abstract

This bachelor's thesis describes simulation of computer networks using open source simulator OMNeT++. We implemented tool, witch generate all important settings for simulation from configuration files of Cisco routers. In this thesis is covered grammar of Cisco IOS. This thesis contains tutorial how to extend this tool with new IOS commands.

Klíčová slova

OMNeT++, IOS gramatika, XML konfigurace směrovače, automatický překlad konfigurace, simulace počítačové sítě

Keywords

OMNeT++, IOS grammar, XML router configuration, automatic translation of configuration, computer network simulation

Citace

Peter Scherfel: Simulace chování sítě na základě analýzy konfiguračních souborů aktivních síťových zařízení, bakalářská práce, Brno, FIT VUT v Brně, 2009

Simulace chování sítě na základě analýzy konfiguračních souborů aktivních síťových zařízení

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Ondřeje Ryšavého Ph.D.

.....
Peter Scherfel
20. května 2009

Poděkování

Ďakujem vedúcemu mojej bakalárskej práce Ing. Ondřeju Ryšavému Ph.D., za jeho odborné rady a časovú dostupnosť pri riešení problémov. Taktiež ďakujem spolupracovníkom projektu ANSA, ktorého súčasťou bola moja bakalárska práca za vytvorenie príjemného pracovného prostredia.

Obsah

Kapitola 1

Úvod

Pri návrhu, rozširovaní, alebo zavádzaní nových služieb je dôležité zaoberať sa otázkou, či sieť bude poskytovať potrebné parametre, ktorými sú hlavne priepustnosť, rýchlosť odozvy a bezpečnosť v závislosti na službách ktoré sieť využívajú.

Pri vytváraní nových sietí väčšinou máme viacej možností ako požadovanú topológiu vytvoriť. Ale v praxi nie je možné realizovať všetky možnosti, podrobiť testom a následne vybrať najvhodnejšiu. Tento postup niekedy ani nie je možný, kedže nemusíme dopredu poznať všetky prekážky (napríklad prostredie). Preto nie je možné pri budovaní, alebo rozširovaní sietí podrobiť sieť testom dopredu. Nakúpiť všetok hardware, otestovať, či vyhovuje požiadavkom a ak náhodou nevyhovuje, vymýšľať iné riešenie je neekonomické. Pri rozširovaní stávajúcich sietí je nevhodné zasahovať do topológie a testovať, lebo by sa mohol narušiť súčasný prevoz.

V týchto prípadoch je vhodnejšie siahnuť po simulačných nástrojoch. Pomocou simulačného nástroju môžme na modelovať budúcu topológiu siete, alebo na základe súčasných konfigurácií ručne, alebo automaticky vytvoriť simuláciu a tú podrobiť testom. Tento prístup k vytváraniu, alebo rozširovaniu sietí je lacný. Veľa simulačných nástrojov disponuje aj grafickým rozhraním, čo nám umožňuje sledovať viacej aspektov siete naraz. Sieť môžme podrobiť rozsiahlejším a viacerím testom. V simulácií môžme pracovať s časovým merítkom. Vďaka týmto možnostiam môžme vytvoriť stabilnejšiu a výkonnejšiu sieť a lepšie optimalizovať do budúcna.

Cieľom tejto práce je vytvoriť nástroj na automatické vytvorenie nastavenia pre simulačný nástroj OMNeT++[5] na základe analýzy konfiguračných súborov Cisco smerovačov. Tento nástroj by mal uľahčiť predovšetkým rozširovanie súčastných sietí, lebo na základe konfiguračných súborov bude jednoduché vytvoriť model siete a sním experimentovať.

1.1 Analýza a preklad konfigurácií

Prvým krokom k vytvoreniu nastavení pre simuláciu je získanie konfigurácií z aktívnych sieťových zariadený a to konkrétne zo smerovačov spoločnosti Cisco. Tieto konfigurácie je možné získať jednoduchým výpisom konfigurácie, alebo nastavenia vyexportovať do externého súboru.

Predtým ako budeme schopný tieto nastavenia analyzovať je potrebné vytvoriť gramatiku konfiguračného súboru, na základe ktorej vytvoríme *lexikálny analyzátor* a *syntaktický analyzátor*, ktorý prevedie syntaktickú kontrolu. Gramatiku príkazov vytvoríme na základe syntaxe príkazov, ktorá je uvedená v dokumentácií k IOS priamo od Cisca [2].

Ďalším krokom je analýza týchto konfigurácií a získanie potrebných parametrov (nastavenie rozhraní, smerovania, ACL), ktoré sú implementované v simulátore OMNeT++. Tieto nastavenia sú prevedené do univerzálnej formy, ktorá slúži na prenos nastavení medzi OMNeT++ a naším nástrojom na preklad. Ako prostredníka sme zvolil XML štruktúru.

XML štruktúra obsahuje len nastavenia aktívnych sieťových zariadený. Ale pre spustenie simulácie je potrebné previesť aj informácie o štruktúre reálnej topológie.

1.2 Simulátor OMNeT++

OMNeT++ je diskrétny objektovo orientovaný simulačný nástroj, ktorý je veľmi flexibilný svojím použitím. Jeho primárne použitie je na simulovanie komunikačných sietí. Pre jeho jeho všeobecnú a flexibilnú architektúru je ho možné použiť aj na simulácie v iných oblastiach ako informačné systémy, siete s frontami, hardware, alebo aj priemyselné systémy.

Hierarchická štruktúra OMNeT++ je zložená z modulov, to znamená, že sa jedná o modulárny systém. Moduly sú na najzákladnejšej úrovni popísané v jazyku *C/C++*. Tieto moduly medzi sebou komunikujú na základe posielania správ a hĺbka zanorenia pri abstrakcií je teoreticky nekonečná, čo umožňuje popísať akúkoľvek štruktúru. Topológia siete je v OMNeT++ popísaná pomocou jazyka *NED*([4]).

OMNeT++ má svoje grafické rozhranie, ktoré nám umožňuje prehľadnejšie sledovať priebeh simulácie a jej výsledky.

1.2.1 Nastavenie simulácie v OMNeT++

Konfigurácia simulácie v OMNeT++ je načítaná vždy z rovnakého súboru a to omentpp.ini. Pokiaľ tento súbor chýba simulácia sa nespustí. Z neho sa načítajú nastavenia simulácie ako napr. ktorá sieť sa ma odsimulovať (simulácia v OMNeT++ môže obsahovať viacej sietí), nastavenia siete, ktoré .ned súbory sa majú načítať a ďalšie. Tieto nastavenia patria medzi všeobecné nastavenia siete.

Nastavenia jednotlivých prvkov (smerovače, prepínače, koncové stanice) sú v súčastnosti načítavané z externých súborov. Tento prístup môže byť nepraktický v prípade, ak chceme vytvoriť simuláciu veľkej siete (napr. veľké množstvo smerovačov).V tomto prípade by vznikol veľký počet konfiguračných súborov, ktorý by bol neprehľadný. Obzvlášť ak si každý smerovač načítava nastavenia rozhraní a nastavenia smerovacej tabuľky z dvoch rozdielnych súborov.

Tento fakt sme sa rozhodli zmeniť a nastavenia všetkých smerovačov v sieťovej topológii budú obsiahnuté v jednom XML dokumente, kde si každý smerovač nájde svoju časť z ktorej bude čerpať nastavenia.

1.3 Prostriedky

Vytváranie nástroja na analýzu a preklad konfiguračných súborov prebiehalo na operačnom systéme Windows XP SP3 s použitím prekladača Cygwin 1.5.25.

Na vygenerovanie lexikálneho a syntaktického analyzátora na základe popisu gramatiky vo forme EBNF (Extended Backus-Naur Form [6]) sme použili nástroj ANTLR verzie 3.1.3(ANother Tool for Language Recognition [1]), ktorého výstupom je C++ zdrojový kód lexikálneho a syntaktického analyzátora. Na preloženie vygenerovaného kódu je potrebné mať nainštalovanú libantlr3c-3.1.3.

Pre vygenerovanie spracovaných nastavení do XML sme použili C++ knižnicu na prácu s XML tinyXML[7] verzie 2.5.3.

Ako zdroj vstupných dát sme použili nastavenia Cisco smerovačov, vygenerované na základe reálneho zapojenia siete.

Kapitola 2

Konfiguračné súbory Cisco smerovačov

IOS je operačný systém Cisco smerovačov a IOS príkazy sú používané na konfiguráciu smerovačov. IOS príkazy a ich syntax vytvárajú takzvaný IOS jazyk. Skupina IOS príkazov vytvára konfiguráciu smerovača.

V tejto kapitole sa budeme zaoberať konfiguračnými súbormi Cisco smerovačov. Podrobne sa pozrieme na ich štruktúru a syntax a podrobne si rozoberieme príkazy, ktoré nás zaujímajú z pohľadu simulácie. To nám pomôže lepšie pochopiť vytváranie gramatiky v kapitole 3.

2.1 Štruktúra konfiguračného súboru

IOS jazyk je deklaratívny jazyk. Deklaratívny jazyk znamená, že jazyk definuje čo sa má urobiť, namiesto toho ako sa to má urobiť. Základným prvkom je *príkaz*. V konfiguračných súboroch môžme rozoznať dva typy príkazov :

- *Jednoduché príkazy* - sú uvedené na jednom riadku a príkaz je ukončený novým riadkom.

  Príklad jednoduchého príkazu :

  ```
  hostname Router1
  ```

- *Stavové príkazy* - príkazy sú uvedené na viacerých riakoch a stavový príkaz mení stav interpreta. Každý stav ma danú množinu príkazov, ktoré prijíma. Tieto príkazy, ktoré interpret prijíma môžu byť jednoduché, alebo stavové.

  Príklad stavového príkazu :

  ```
  interface FastEthernet0/0
    ip address 192.168.1.1 255.255.255.0
    ip access-group ACL-10 in
    duplex auto
    speed auto
  !
  ```

Ďalšiou súčasťou konfiguračného súboru sú jednoriadkové poznámky, ktoré uvádzajú jednoriadkové komentáre, alebo ukončujú stavové príkazy. Tieto poznámky vždy začínajú znakom `!`.

2.2 Potrebné príkazy pre simuláciu

V následujúcich podkapitolách postupne prejdeme a rozoberieme príkazy, ktoré sú potrebné pre vytvorenie simulácie. Ide o príkazy, ktoré definujú nastavenia smerovačov v simulovanej topológii.

Medzi nastavenia, ktoré vieme previesť a odsimulovať v simulácií patria nastavenia pre hostname, interface, smerovanie pre OSPF, RIP a statické smerovanie a ACL.

2.2.1 Nastavenie pomenovania

Je to najjednoduchší príkaz. Ide o jednoriadkový príkaz, ktorý preberá len jeden argument a to meno, ktoré smerovač použije ako hostname. Tento príkaz slúži hlavne k sprehľadneniu topológie. Hostname sa nedá použiť pre jednoznačnú identifikáciu smerovačou v seti. Preto pre identifikáciu smerovačou použijeme ID smerovača, čo je najvyšia IP adresa nastavená na smerovači. Toto nastavenie bude slúžiť v simulácií k lepšej orientácií, ktorý smerovač v simulovanej topológii odpovedá ktorému smerovaču v reálnej topológií.

Syntax príkazu hostname :

```
hostname name
```

2.2.2 Nastavenie rozhrania

Príkaz interface patrí medzi najzákladnejšie a najdôležitejšie nastavenia Cisco smerovačov. Vytvára nastavenie pre jednotlivé rozhrania ako IP adresa, šírka prenosového pásma a podobne. Ide o stavový príkaz, to znamená príkaz zmení stav interpreta a prijíma iné príkazy. Sú to príkazy, ktoré vytvárajú konkrétne nastavenia pre rozhranie.

Na Cisco smerovačoch môžme rozoznať niekoľko typov rozhraní.

- *Vytáčané rozhranie* - ISDN,
- *LAN rozhranie* - Ethernet, FastEthernet, GigabitEthernet, Token Ring
- *Sériové rozhranie* - vysoko rýchlostné sériové rozhrania (HSSI), synchronne sériové rozhrania (PPP)
- *Logické a virtuálne rozhrania* - loopback, tunel, NULL rozhranie

Z vyššie uvedených rozhraní nás pre nastavenie simulácie budú zaujímať len LAN a sériové rozhrania. Problém, ktorý bude treba riešiť pri preklade konfiguračných súborov, je pomenovanie rozhraní. V IOS sú rozhrania pomenované menom (*FastEthernet, Serial*) a číslom v tvare X/X, poprípade X/X/X. Tieto čísla určujú číslo slotu a portu. V simulátore OMNeT++ sú rozhrania pomenované menom (*eth, ppp*) a číslom. Toto číslo slúži ako jednoznačný identifikátor rozhrania na smerovači. Čísla sú v rozmedzí 0-16.

Syntax príkazu interface je nasledovná :

```
interface typ číslo [logické meno]
```

Príkaz interface, len zmení mód interpreta a po jeho zadaní je pripravený prijímať príkazy, ktoré definujú nastavenie rozhrania. Príkazy, ktoré vieme previesť do simulácie sú následovné :

- `ip address`
- `bandwidth`
- `duplex`
- `clock rate`
- `shutdown`

2.2.3 Nastavenie smerovania

Smerovanie je dôležitým prvkom pri vytváraní topológii. Preto je potrebné zahrnúť nastavenie smerovania a preniesť ho do simulácie. V OMNeT++ sú implementované statické cesty a smerovacie protokoly RIP a OSPF.

Statické smerovanie

Statické smerovanie je podporované simulátorom OMNeT++, ale nie všetky jeho parametre sú podporované. Statické cesty definujú cieľovú sieť a adresu ďalšieho smerovača, ktorému sa má paket poslať, alebo názov výstupného rozhrania, ktorým sa ma paket poslať. Tu vzniká problém, ktorý sme si popísali v kapitole 2.2.2 a to s pomenovaním rozhraní, ktoré bude potreba previesť na názvy, ktoré podporuje simulátor OMNet++ .

Syntax príkazu `ip route` je následovná :

```
ip route network mask {ip-address | interface-type interface-number [ip-address]}
[dhcp] [distance] [name next-hop-name] [permanent | track number] [tag tag]
```

Smerovanie pomocou RIP

Smerovací protokol RIP je typu *distance-vector*. Na Cisco smerovačoch rozoznávame dve verzie protokolu RIP. RIPv1 a RIPv2. Do konfigurácie obidvoch verzií sa v IOS vstupuje rovnakým spôsobom a to cez príkaz `router rip`.

Simulátor OMNeT++ však podporuje len protokol RIPv1. Obidve verzie sa konfigurujú rovnakým spôsobom a to o ktorú verziu ide určuje až príkaz `version`. Tento problém je bližšie popísaný v kapitole 2.3. Ak by sme príkaz `version` zanedbali, tak by simulácia neodpovedala skutočnosti, lebo v simulácií by sa vyskytoval smerovací protokol RIPv1, ale v skutočnosti by bol nastavený RIPv2.

Príkaz ktorý slúži pre konfiguráciu smerovacieho protokolu RIPv1 a RIPv2 :

```
router rip
```

Syntax príkazov ktoré natávajú parametere protokolu RIP a sú implementované v OMNeT++:

```
network network-number
passive-interface [default] {interface-type interface-number}
redistribute protocol [process-id] [metric {metric-value | transparent}]
```

Smerovanie pomocou OSPF

Smerovací protokol OSPF je typu *link-state*. OSPF protokol je implementovaný v simulátore OMNeT++, ale nie sú podporované jeho všetky nastavenia. Pri nastaveniach OSPF sa stretávame s faktom, že nastavenia OSPF v konfiguračnom súbore môžu byť uvedené na viacerých miestach. Napríklad :

```
interface FastEthernet0/0
 ip address 192.168.1.1 255.255.255.0
 ip ospf hello-interval 200
 ip ospf priority 2
 duplex full
 speed auto
!
router ospf 1
 network 192.168.2.0 0.0.0.255 area 3
!
```

Pri prevádzaní nastavenia OSPF do simulácie bude potreba v niektorých prípadoch zmeniť označenie oblasti v príkaze `network`, tak aby bolo vhodné pre OMNeT++. V cisco smerovačoch je oblasť označená ako ako 32 bitové číslo, alebo ako IP adresa. V simulátore OMNeT++ je pre označenie oblasti použitý len formát IP adresy.

Syntax najdôležitejších príkazov :

```
network address wildcard-mask area area-id
```

2.2.4 Access control list

V simulátore OMNet++ sú podporované viaceré typy ACL. Sú to ACL typu *standard, extended* a *named*. ACL podporuje protokoly *ip, tcp, udp, icmp*. ACL *standard* a *extended* sú jednoriadkové príkazy a ACL *named* patrí medzi stavové príkazy.

ACL typu *standard* majú najjednoduchšiu syntax:

```
access-list acl-number {permit|deny} source [source-wildcart] [log]
```

Pri ACL typu *extended* je situácia mierne zložitejšia. *Extended* ACL podporujú rôzne protokoly (*ip, tcp, udp, icmp*) a každý z týchto protokolov ma odlišnú syntax. Uvedieme si syntax *extended* ACL podľa typu protokolu, ktorý podporujú.

- *ip*

  ```
  access-list acl-number [dynamic dynamic-name [timeout minutes]] {permit|deny}
  protocol source source-wildcard destination destination-wildcard
  [precedence precedence] [tos tos] [log | log-input]
  [time-range time-range-name]
  ```

- *icmp*

  ```
  access-list acl-number [dynamic dynamic-name [timeout minutes]] {permit|deny}
  icmp source source-wildcard destination destination-wildcard
  [icmp-type] [[icmp-type icmp-code] | [icmp-message]]
  [precedence precedence] [tos tos] [log | log-input]
  [time-range time-range-name]
  ```

- *tcp*

```
access-list acl-number [dynamic dynamic-name [timeout minutes]] {deny|permit}
tcp source source-wildcard [operator [port]] destination destination-wildcard
[operator [port]] [established] [precedence precedence] [tos tos]
[log | log-input] [time-range time-range-name]
```

- *udp*

```
access-list acl-number [dynamic dynamic-name [timeout minutes]] {deny|permit}
tcp source source-wildcard [operator [port]] destination destination-wildcard
[operator [port]] [precedence precedence] [tos tos] [log | log-input]
[time-range time-range-name]
```

2.3 Problém prevoditeľnosti

Pri vytváraní nastavení pre simuláciu z konfiguračných súborov, sa stretávame s problémom, že simulátor plne neimplementuje funkcionalitu Cisco smerovačov. Tento problém uvediem na príklade nastavení ACL.

Majme dva ACL záznamy.

```
access-list 101 permit tcp host 192.168.4.1 eq www any
access-list 101 permit tcp host 192.168.4.1 eq www any established
```

Simulátor OMNet++ neimplementuje parameter established. ACL pravidla, ktoré uvedené v príklade sú rozdielne. Ale ak by gramatika nezohľadnila čo všetko OMNet++ podporuje, pri vytváraní nastavenia pre simuláciu by z vyššie uvedených pravidiel, mohlo vzniknúť pravidlo s rovnakou funkcionalitou.

Tento problém sa týka nastavení rozhraní, smerovania a ACL. Pri ignorovaní tohto problému by mohla vzniknúť chyba pri preklade konfiguračných súborov, dôsledkom ktorej by simulácia nemusela odpovedať nastaveniam skutočných smerovačov. Pri riešení tohto problému, je nutné sa zamyslieť ktoré z neimplementovaných nastavení v simulátore OMNet++ skutočne menia funkcionalitu (napríklad parameter `log` pri ACL).

Riešenie tohto problému si uvedieme v kapitole 3, keď sa budeme zaoberať vytváraním gramatiky.

2.4 Príklad konfiguračného súboru

Ukážeme si príklad konfiguračného súboru Cisco smerovača pre lepšie pochopenie syntaxy príkazov, ktoré sme si v predchádzajúcich podkapitolách uviedli.

```
!
version 12.4
no service password-encryption
!
hostname Router
!
interface FastEthernet0/1
 ip access-group mojeACL in
```

```
 ip address 192.168.2.1 255.255.255.0
 duplex auto
 speed auto
 shutdown
!
access-list 10 permit 192.168.1.0 0.0.0.255
ip access-list extended mojeEXTENDED
 permit ip host 192.168.1.1 any
 deny ip 192.168.1.0 0.0.0.255 any
!
router ospf 1
 network 192.168.1.0 0.0.0.255 area 192.168.1.1
 network 192.168.2.0 0.0.0.255 area 3
!
router rip
 network 192.168.1.2
 passive-interface FastEthernet0/0
 redistribute ospf 1 metric 4
!
line con 0
line vty 0 4
 login
!
end
```

2.5 Zhrnutie

V tejto kapitole sme si uviedli príkazy nastavení, ktoré sme schopný v simulátore OMNeT++ odsimulovať. Ukázali sme si ako vyzerá ich syntax, ktorá bude potrebná pri vytváraní gramatiky.

Medzi tieto nastavenia patria hostname, interface, RIP, OSPF, statické smerovanie, ACL. Je dôležité brať v úvahu fakt, že simulátor OMNeT++ nepodporuje všetky nastavenia Cisco smerovača, preto je nutné v gramatike nejakým spôsobom zahrnúť aj nastavenia, ktoré nie sú podporované. Ak by sme tento problém zanedbali po preklade konfiguračného súboru by mohlo vzniknúť nastavenie pre simuláciu, ktoré neodpovedá skutočnosti.

Kapitola 3

Jazyk a gramatika konfiguračného súboru

V tejto kapitole si rozoberieme a zadefinujeme gramatiku konfiguračného súboru Cisco smerovača.

V prvej časti kapitoly si ozrejmíme termíny potrebné k všeobecnej definícií gramatiky a vytváraniu gramatiky ako *terminál, neterminál, gramatiky.*

V druhej časti kapitoly si vysvetlíme postup vytvárania gramatiky na základe syntaxe príkazov, ktoré sme si uviedli v kapitole 2.

3.1 Gramatika

V tejto podkapitole si najprv zadefinujeme termíny, ktoré sú potrebné k utvoreniu formálnej definícií gramatiky.

3.1.1 Terminálne a neterminálne symboly

Terminálne a neterminálne symboly, sú symboly, ktoré sú použité na tvorbu pravidiel v gramatike.

- *Terminálne symboly* sú reťazce, ktoré reprezentujú vstup pre gramatiku a nemôžu byť rozložené na menšie logické jednotky, bez toho aby stratili svoj význam.
- *Neterminálne symboly* vznikajú ako výsledok uplatnenia odvodzovacieho pravidla gramatiky. Skladajú sa zo sekvencie terminálnych, alebo neterminálnych symbolov. Neterminálne symboly, môžu obsahovať rekurziu.

Pre lepšie pochopenie čo sú terminálne a neterminálne symboly si uvedieme príklad. Gramaticky zadefinujeme číslo typu `integer`:

```
integer = ['-'], digit {digit};
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
```

V tomto príklade medzi terminály patria `(-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)` a medzi neterminály patrí `integer` a `digit`.

3.1.2 Definícia gramatiky

Gramatika je súbor pravidiel popisujúcich, ktorý reťazec poskladaný z abecedy jazyka je syntakticky správny v rámci jazyka. Gramatika určuje len pozíciu a manipuláciu s reťazcami jazyka. Nedefinuje nič ďalšie o jazyku, ako napríklad sémantiku (čo presne znamená konkrétny reťazec).

Gramatika G je štvorica (N, Σ, P, S), kde :

- N je konečná množina *neterminálnych* symbolov.
- Σ je konečná množina *terminálnych* symbolov, kde $N \cap \Sigma = \varnothing$
- P je konečná množina *odvodzovacích* pravidiel kde pravidlá sú tvaru :
  $(\Sigma \cup N)^* N (\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$
- S je prvok z množiny N nazývaný ako počiatočný symbol.

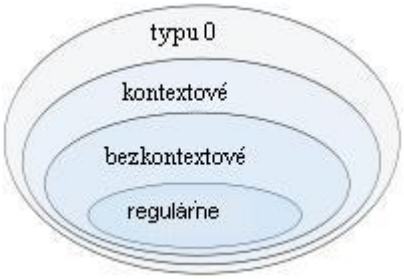
Príklad gramatiky si uvedieme až v ďalšej podkapitole, keď budeme vytvárať gramatiku pre konfiguračné súbory Cisco smerovačov.

3.1.3 Typy gramatik - Chomského hierarchia

Chomského hierarchia je hierarchia tried gramatík, ktoré generujú jazyky. *Chomského hierarchia* bola vytvorená Noamem Chomskym v roku 1956 a rozdeľuje gramatiky do nasledujúcich typov :

- **typ 0** - Zahrňuje všetky jazyky. Generuje jazyky, ktoré sú rozpoznateľné pomocou *Turingového stroja*. Tieto gramatiky sa nazývajú aj *rekurzívne spočetné gramatiky*.
- **typ 1** (kontextové gramatiky) - Tieto gramatiky sa skladajú z pravidiel typu :
  $\alpha A \beta \longrightarrow \alpha \gamma \beta$
  kde A je neterminálny symbol a α, β, γ sú reťazce terminálov a neterminálov. α *a* β môžu byť prázdne, ale γ musí byť neprázdne. α *a* β tvoria kontext, ktorý určuje, či A ide prepísať na γ. Jazyky generované týmto typom gramatík, sú rozpoznávané pomocou *Lineárne ohraničených automatov*.
- **typ 2** (bezkontextové gramatiky) - Tieto gramatiky sa skladajú z pravidiel typu :
  $A \longrightarrow \gamma$
  kde A je neterminálny symbol a γ je reťazec terminálov a neterminálov. Jazyky generované týmito gramatikami sú rozpoznávané pomocou *zásobníkových automatov*.
- **typ 3** (regulárne gramatiky) - Tieto gramatiky generujú *regulárne jazyky*. Skladajú sa z pravidiel, kde na ľavej strane je jeden neterminálny symbol a pravá strana sa skladá z reťazca terminálov, ktorý môže byť následovaný práve jedným neterminálom. Jazyky generované týmito gramatikami sú rozpoznávané pomocou *konečných automatov*.

Na obrázku 3.1 je znázornení vzťah medzi triedami gramatík *Chromského hierarchie*



Obrázek 3.1: Vzťah medzi gramatikami podľa *Chromského hierarchie*

3.2 IOS jazyk a jeho gramatika

V tejto podkapitole si prejdeme a ukážeme niektoré vlastnosti IOS jazyka, ktorých pochopenie nám uľahčí vytváranie gramatiky.

Predtým ako začneme túto kapitolu je potrebné zaviesť pojem *token*, pretože v spojitosti s gramatikou ho budeme často používať.

3.2.1 Token

Token je kategorizovaný ako blok textu. Token sa môže skladať z akýchkoľvek znakov, ktoré sú súčasťou štruktúrovaného textu. Najčastejšie sú definované pomocou *regulárnych výrazov*, ktoré je schopný prečítať lexikálny analyzátor. Lexikálny analyzátor číta znaky zo vstupného textu a vytvára z nich tokeny. Tento proces sa nazýva *tokenizácia*, kedy lexikálny analyzátor priradí blokom textu význam (tokenu `+` môže byť priradený význam ako `operator`).

3.2.2 Vlastnosti jazyka IOS

Skôr ako začneme vytvárať gramatiku pre konfiguračný súbor Cisco smerovača si uvedieme niektoré vlastnosti tohto jazyka, ktoré treba brať v úvahu pri vytváraní gramatiky. Hlavne sa zameriame na príkazy nastavení, ktoré vieme previesť do simulátoru OMNeT++.

Syntax príkazov

Pri syntaxy príkazov sa stretávame s problémom a to, že syntax nie je možné jednotne zadefinovať. Každý príkaz ma svoj vlastný formát a syntax. Väčšina príkazov je nasledovaná dvojicami kľúčové slovo/hodnota. Z tohto vyplýva, že pre väčšinu príkazov budú vytvorené odvodzovacie pravidlá zvlášť.

Typy príkazov

V jazyku IOS rozoznávame dva typy príkazov ako sme si uviedli v kapitole 2. Rozdiel medzi týmito dvoma typmi je hlavne v tom, že jednoriadkové príkazy nemenia stav interpreta a stavové menia. To znamená, že po zmene stavu interpreta, interpret prijíma, len určitú množinu príkazov. Ktoré to sú určíme z manuálových stránok spoločnosti Cisco [2]. Ak by sa v tejto sekcií vyskytol príkaz, ktorý nie je prípustný v danom stave interpreta gramatika to musí rozpoznať a syntaktický analyzátor ohlásiť chybu syntaxe.

Kľúčové slová ako identifikátory

Jazyk IOS umožňuje použiť klúčove slovo ako identifikátor. Napríklad v názve ACL sa môže objaviť príkaz `interface`. V tomto prípade však nemôže byť posudzovaný ako príkaz nastavujúci rozhranie, ale musí byť posúdený ako reťazec, ktorý označuje ACL. Túto možnosť je nutné zohľadniť v gramatike, ktorú budeme vytvárať.

Počet tokenov na rozoznanie príkazu

V jazyku IOS existujú príkazy, ktoré nie je možné rozpoznať na základe jedného tokenu. Niektoré príkazy (najčastejšie `ip` a `router`) je možné rozoznať až na základe ďalšieho tokenu, alebo viacerých následujúcich.

3.2.3 Gramatika jazyka IOS

Pri vytváraní gramatiky vychádzame z analýzy syntaxe príkazov na základe dokumentácie[2]. Ako sme si v predchádzajúcej kapitole uviedli vlastnosť, že každý príkaz má svoju vlastnú syntax, budeme vo väčšine prípadov vytvárať odvodzovacie pravidlá zvlášť pre každý príkaz. Týmto spôsobom môžme odvodzovacie pravidlá rozdeliť do dvoch kategórií :

- pravidlá pre jednoriadkové príkazy
- pravidlá pre stavové príkazy

Toto rozdelenie nám pomôže postaviť prehľadnejšiu gramatiku.

Predtým ako začneme vytvárať gramatiku, je potrebné zaviesť štandard, akým spôsobom budeme popisovať gramatiku. Pre tento účel si zvolíme EBNF (Extended Backus-Naur Form). Ide rozšírenú formu BNF (Backus-Naur Form [6]), hlavne o možnosť v jednom výraze vyjadriť opakovanie symbolu a voliteľného symbolu, čo podstatne sprehľadní popis gramatiky.

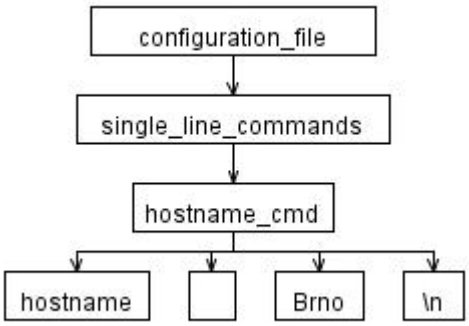
Výhodou použitia EBNF je, že umožňuje definovať terminálne symboly, aj neterminálne symboly. Čo v praxy znamená pravidla pre lexikálny analyzátor, ktorý spracúva textové znaky na vstupe a podľa definície terminálnych symbolov ich kategorizuje do tokenov a pravidla pre syntaktický analyzátor, ktorý spracúva tokeny od lexikálneho analyzátora a na základe definície neterminálnych symbolov vytvára abstraktný syntaktický strom.

Vytváranie gramatiky si ukážeme na dvoch príkladoch. Prvý bude jednoduchší, a to vytváranie pravidiel pre jednoriadkové príkazy a druhý bude vytváranie pravidiel pre stavové príkazy. Všetky ostatné príkazy sa budú vytvárať analogicky. Predtým ako začneme vytvárať pravidlá pre jednotlivé príkazy, je potrebné si pripraviť základ, aby výsledná gramatika bola prehľadnejšia a rozširovanie gramatiky o nové príkazy bolo jednoduché.

Ako sme si uviedli v definícií gramatiky (3.1.2), každá gramatika musí obsahovať štartovacie pravidlo, od ktorého sa budú odvýjať všetky ostatné odvodzovacie pravidla. V prvom kroku, si vytvoríme štartovacie pravidlo, pravidlá ktoré budú zahŕňať pravidlá pre jednoriadkové príkazy a pre stavové príkazy a popis celkovej štruktúry konfiguračného súboru.

```
configuration_file = {(comment | single_line_commands | mode_commands)},
                     END_CONFIGURATION, EOF;
comment = EXCLAMATION_MARK, {WS | STRING}, NEW_LINE;

single_line_commands = ;
mode_commands = ;
```



Obrázek 3.2: Postup pravidiel pri generovaní príkazu `hostname Brno`.

```
EXCLAMATION_MARK = '!';
END_CONFIGURATION = 'end', [WS], NEW_LINE;
NEW_LINE = '\n' | '\r' | '\r\n';
WS = ('\t' | ' '), {('\t' | ' ')};
STRING = ANY_CHAR, {ANY_CHAR};
ANY_CHAR = -(NEW_LINE | WS)*;
```

Vytvorili sme základnú štruktúru konfiguračného súboru a pripravili sme si pravidlá (`single_line_comman` `mode_commands`), ktoré neskôr budú obsahovať odvodzovacie pravidlá konkrétnych príkazov IOS jazyka.

Základ gramatiky je hotový a nachystaný. Teraz doňho môžme začať pridávať pravidlá, ktoré budú reprezentovať jednotlivé príkazy IOS jazyka. Ako prvý príkaz o ktorý gramatiku rozšírime, bude príkaz `hostname`. Je to veľmi jednoduchý príkaz ktorý má len jeden parameter a to názov, akým ma byť smerovač pomenovaný.

```
single_commands = hostname_cmd;
hostname_cmd = HOSTNAME, WS, STRING, [WS], NEW_LINE;

(*Zadefinujeme kľúčové slovo.*)
HOSTNAME = 'hostname';
```

Na obrázku 5.1 je vidieť postupnosť pravidiel, akou je možné na základe gramatiky vygenerovať príkaz `hostname brno`.

Teraz si gramatiku rozšírime o stavový príkaz `interface`. Je treba mať na pamäti, že príkaz `interface` zmení stav IOS interpretu a po ňom následujú príkazy, ktoré ho upresňujú (`interface` vstúpi do nastavení rozhrania a ďalšie príkazy nastavujú rozhranie). Tieto príkazy sú jednoriadkové, ale nemôžu byť zahrnuté do pravidla `single_commands`, lebo nie sú očakávané základom stave interpreta, ale v stave, ktorý uviedol prvý príkaz stavového príkazu. Preto tieto príkazy môžme zhrnúť do bloku a tak k nim budeme pristupovať aj v gramatike. Pri vytváraní pravidiel pre stavový príkaz *interface* budeme uvažovať len nastavenia `ip address, speed, bandwidth`. Ostatné sa vytvoria analogicky z ich syntaxe.

```
mode_commands = interface_cmd;
interface_cmd = INTERFACE, WS, STRING, [WS], NEW_LINE,
```

```
                interface_cfg_block, WS, EXCLAMATION_MARK, [WS], NEW_LINE;
interface_cfg_block = {(interface_ip_cfg | interface_speed_cfg |
                      interface_bandwidth_cfg)};
interface_ip_cfg = IP, WS, ADDRESS, WS, IP_ADDR, WS, IP_ADDR, [WS], NEW_LINE;
interface_speed_cfg = SPEED, WS, SPEED_VALUE, [WS], NEW_LINE;
interface_bandwidth_cfg =    BANDWIDTH, WS, NUMBER, [WS], NEW_LINE;

(*Zadefinujeme kľúčové slová.*)
INTERFACE = 'interface';
IP = 'ip';
ADDRESS =    'address';
SPEED = 'speed';
BANDWIDTH = 'bandwidth';

SPEED_VALUE = 'auto' | '10' | '100';
NUMBER = DIGIT,{DIGIT};
DIGIT = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
(*Definícia terminálu IP_ADDR je obsiahla a kôli prehľadnosti je uvedená
až v prílohe, kde je znázornená celá gramatika.*)
```

Na obrázku 3.3 je vidieť postupnosť pravidiel, akou je možné na základe gramatiky vygenerovať príkaz :

```
interface FastEthernet0/0
 ip address 192.168.1.1 255.255.255.0
 speed auto
 bandwidth 64
!
```

Na dvoch príkladoch sme si ukázali, ako sme postupovali pri vytváraní gramatiky. Ako vyzera spojenie týchto dvoch príkladov do celej gramatiky, je vidieť v prílohe.

V časti 2.3 sme si vysvetlili problém prevoditeľnosti. Tento problém budeme riešiť pravidlom `unsupported`, ktoré bude obsahovať všetky ostatné alternatívy k definovaným pravidlám. Pravidlo bude zahrnuté ako alternatíva k pravidlu, ktoré definuje príkaz, ktorý nie je plne podporovaný v simulátore OMNeT++. Toto pravidlo, bude reprezentovať parametre, ktoré nie sú implementované.

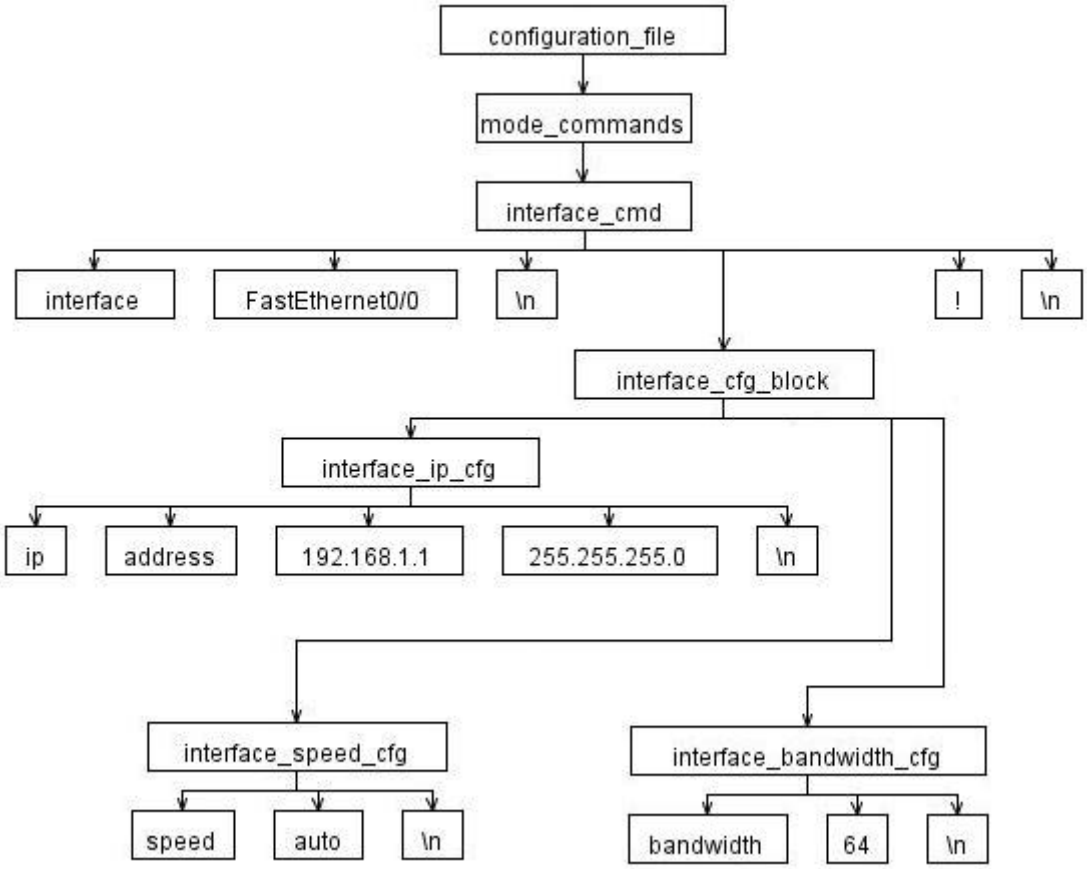
```
unsupported = ~(single_command | mode_commands);
```

Pravidlo `interface_cmd` rozšírené o parametre, ktoré nepodporuje OMNeT++ bude vyzerať následovne.

```
interface_cmd = INTERFACE, WS+, STRING, WS*, NEW_LINE,
                interface_cfg_block, WS+, EXCLAMATION_MARK, WS+, NEW_LINE;
interface_cfg_block = (interface_ip_cfg | interface_speed_cfg |
                      interface_bandwidth_cfg | unsupported)*;
```

Pomocou pravidla `unsupported` môžme pristupovať k parametrom, alebo príkazom, kroré nie sú implementované v OMNeT++ a pri ich výskyte príslušne zareagovať. Viacej si o tom povieme v kapitole 5.3, kde sa budeme zaoberať pridávaním akcií k pravidlám za pomoci nástroja ANTLR.

Celá gramatika je uvedená v prílohe D.



Obrázek 3.3: Postup pravidiel pri generovaní príkazu `interface FastEthernet0/0`.

3.3 Zhrnutie

V prvej časti tejto kapitoly, sme si vysvetlili základné termíny ako terminálny a neterminálny symbol. Uviedli sme formálnu definíciu gramatiky a ukázali, ako sa gramatiky rozdeľujú podľa Chomského hierarchie.

V druhej časti sme uviedli vlastnosti jazyka IOS na ktoré je dôležité brať ohľad pri vytváraní gramatiky. Ako prvý krok vytvárania gramatiky sme vytvorili základnú štruktúru konfiguračného súboru Cisco smerovača. V nej sme pomocou EBNF zadefinovali základné terminálne symboly, vytvorili pravidlá pre komentáre a vytvorili dve pravidlá ktoré budú obsahovať pravidlá pre odvodzovanie príkazov, kde rozoznávame dva druhy príkazov. Toto rozdelenie zprehľadní gramatiku a v budúcnosti nám umožní jednoduché rozširovanie gramatiku o podporu nových príkazov jazyka IOS.

Kapitola 4

Nastavenia pre simuláciu

V tejto kapitole si vysvetlíme, akým spôsobom budeme prenášať nastavenia jednotlivých smerovačov a čo všetko je potrebné nato, aby sme vytvorili simuláciu danej siete.

Uvedieme si ktoré nastavenia z Cisco smerovačov sú validné pre simulátor OMNeT++ a ktoré je potrebné nejakým spôsobom transformovať, aby sme ich boli schopný previesť do simulácie.

4.1 Nastavenia smerovačov

Ako sme si uviedli v kapitole 2, nastavenia Cisco smerovačou sú uložené v konfiguračných súboroch. V kapitole 3 sme vytvorili gramatiku, ktorá popisuje tieto konfiguračné súbory. Avšak simulátor OMNeT++ nie je schopný tieto konfiguračné súbory prečítať, preto je potrebné previesť tieto nastavenia do formátu, ktorý je univerzálny a čitateľný pre OMNeT++.

4.1.1 XML ako prostredník

Ako univerzálny formát uloženia nastavení Cisco smerovačov, použijeme formát XML (Extensible Markup Language). XML je značkovací jazyk, ktorý umožňuje popísať štruktúru dokumentu z hľadiska vecného obsahu. Týmto spôsobom je možné vytvárať dokumenty, ktoré sú univerzálne čitelné a preto sme ako prostredníka na uloženie nastavení smerovačou zvolili práve XML.

Nastavenie všetkých smerovačov z reálnej sieti budeme ukladať do jedného XML dokumentu, preto koreňový element, bude obsahovať elementy, ktorý každý jeden reprezentuje jeden smerovač.

Identifikácia smerovačov

Vzhľadom k tomu, že informácie o nastavení všetkých smerovačov uchovávame v jednom XML dokumente, je potrebné každý smerovač (element) označiť jednoznačným identifikátorom. Ako tento identifikátor nemôžme použiť pomenovanie zadané pomocou príkazu `hostname`, lebo toto označenie nemusí byť jedinečné. Môže sa stať, že všetky smerovače majú nastavený rovnaký hostname. Preto ako identifikátor je potrebné zvoliť údaj, ktorý je jedinečný v rámci celej siete. Pre tento účel nám poslúži IP adresa, ktorá je nastavené na rozhraní smerovača. Tento identifikátor sa môže stať nepoužiteľný, pokiaľ sa v topológií vyskytne NAT (Network Address Translation). NAT môže spôsobiť, že v topológií sa vyskytnú dve rovnaké IP adresy.

Pre jednoznačnú identifikáciu smerovača v XML dokumente zvolíme najvyššiu IP adresu, ktorá je nastavená na rozhraniach smerovača.

Štruktúra XML dokumentu

Nastavenia v XML dokumente pre jeden konkrétny smerovač rozdelíme do troch hlavný skupín :

- nastavenia rozhraní
- nastavenia smerovania
- nastavenia ACL

4.1.2 Nastavenia rozhraní

Informácie o nastaveniach rozhraní sú uložené všetky spolu pod elementom `Interfaces`, ktorého umiestnenie sa v XML dokumente sa dá vyjadriť pomocou *XPath* ako

```
/Routers/Router[@id='router_id']/Interfaces
```

a nastavenia jednotlyvých rozhraní sú pod elementom `Interface` s atribútom `name`, ktorý označuje konkrétne rozhranie na smerovači a slúži aj ako identifikátor rozhrania v rámci jedneho smerovača. Umiestne nastavení jednotlyvých rozhraní pomocou *XPath* :

```
/Routers/Router[@id='router_id']/Interfaces/Interface[@name='interface_name']
```

Informácie o nastaveniach rozhraní smerovača nie sú všetky prenositeľné do simulácie a je potrebné ich upraviť. Ide o nastavenia pomenovaní rozhraní, ktoré slúžia aj ako identifikácia rozhraní. Ako sme si uviedli v kapitole 2, OMNeT++ označuje rozhrania iným spôsobom, ako sú označované v konfiguračných súboroch Cisco smerovačov. Preto je potrebné Ethernetové rozhrania premenovať na *eth0-16* a sériové na *ppp0-16*.

Smerovače obsiahnuté v simulácií nemôžu obsahovať rozhrania, ktoré nie sú spojené so žiadnym iným rozhraním. Túto možnosť simulátor OMNeT++ nepodporuje, preto je potrebné rozhrania, ktoré sú vypnuté pomocou príkazu `shutdown` odstrániť z nastavenia smerovača.

4.1.3 Nastavenia smerovania

Nastavenia smerovacích protokolov je uchovávané pod elementom `Routing`. Umiestnenie tohto elementu je následujúce :

```
/Routers/Router[@id='router_id']/Routing
```

Nastavenia pre jednotlivé protokoly, sú uchovávané pod týmto elementom, označený každý menom protokolu, pre ktorý uchováva nastavenia.

Pri nastavení smerovania je potrebné zmeniť označenie oblasti pri OSPF protokole. Cisco smerovače umožňujú zadať označenie oblasti pomocou 32 bitového čísla vo formáte IP adresy, alebo dekadickom formáte. V simulátore OMNeT++ je možné oblasť v OSPF označiť len pomocou 32 bitového čísla vo formáte IP adresy. To znamená pre korektné prevedenie nastavenia OSPF, je potrebné označenie oblasti, ak je zadané číslom, previesť na IP adresu. Tento prevod si ukážeme na príklade, ak máme oblasť označenú napr. číslom 358.

```
358 - 00000000 00000000 00000001 01100110
          0    .    0    .    1    .   102
```

Oblasť s číslom 358 v Cisco smerovači, bude v simulátore OMNeT++ označená ako 0.0.1.102.

4.1.4 Nastavenia ACL

Nastavenia pre ACL sú uchovávané taktiež v zvlášť sekcií. Táto sekcia je označená elementom ACLs, kde každá skupina pravidiel, je zvlášť označená názvom, alebo číslom. Zaradenie nastavení ACL v XML dokumente je následovné.

```
/Routers/Router[@id='router_id']/ACLs
```

A konkrétne ACL pravidlo ma XPath

```
/Routers/Router[@id='router_id']/ACLs/ACL[@no='name']/entry
```

Pri nastaveniach ACL nie sú informácie, ktoré by trebalo nejakým spôsobom transformovať. Všetky nastavenia pre ACL typu *standard*, *extended* a *named* uchovávame v jednotnej XML štruktúre štruktúre.

Nastavenia pre ACL typu *standard* sa dajú uložiť ako nastavenia pre *extended*, len je potreba zaviesť defaultné nastavenia, ktoré v podstate ACL typu *standard* prevedú na *extended*. Napríklad ACL

```
ip access-list standard natACL
 permit 192.168.1.0 0.0.0.255
```

je možné zapísať ako

```
ip access-list extended natACL
 permit ip 192.168.1.0 0.0.0.255 any
```

Preto pre uchovanie nastavení ACL typu *standard* je potrebné pridať informáciu o ciely, do ktorého je paket určený. Aké hodnoty pre zdroj a destináciu použijeme pri ukladaní nastavení ACL nám určuje tabuľka 4.1.

Zdroj/cieľ	**IP**	**Maska**
any	0.0.0.0	255.255.255.255
host	X.X.X.X	0.0.0.0

Tabulka 4.1: Hodnoty zdroj/cieľ pri nastaveniach pre ACL.

Podľa tabuľky 4.1 ACL pravidlo, ktoré sme si vyššie uviedli, bude v XML dokumente s nastaveniami uložené ako *extended*, kde adresa cieľa bude `0.0.0.0` s maskou `255.255.255.255` a protokol `ip`. Tieto hodnoty musia byť pridané ku každému ACL *standard* pravidlu, aby bolo možné nastavenia ACL pravidiel uchovávať v jednotnej štruktúre.

4.1.5 Príklad nastavení v XML

Ukážeme si príklad nastavenia Cisco smerovača uloženého v XML štruktúre.

```
<?xml version='1.0' ?>
<Routers>
    <Router id='0.0.0.0'>
        <Hostname>router</Hostname>
        <Interfaces>
            <Interface name='eth1'>
                <IPAddress>192.168.10.1</IPAddress>
                <Mask>255.255.255.0</Mask>
                <Duplex>auto</Duplex>
                <Speed>auto</Speed>
            </Interface>
        </Interfaces>
        <ACLs>
            <ACL no='mojeACL'>
                <interfaces>
                    <interface dir='in'>FastEthernet0/0</interface>
                </interfaces>
                <entry>
                    <action>permit</action>
                    <protocol>tcp</protocol>
                    <IP_src>192.168.4.1</IP_src>
                    <WC_src>0.0.0.0</WC_src>
                    <port_op_src>eq</port_op_src>
                    <port_beg_src>www</port_beg_src>
                    <IP_dst>0.0.0.0</IP_dst>
                    <WC_dst>255.255.255.255</WC_dst>
                    <port_op_dst>range</port_op_dst>
                    <port_beg_dst>ftp</port_beg_dst>
                    <port_end_dst>50</port_end_dst>
                </entry&amp;amp;amp;gt;
            </ACL>
        </ACLs>
        <Routing>
            <Ospf>
                <RFC1583Compatible />
                <Areas>
                    <Area id='0.0.0.0'>
                        <Networks>
                            <Network>
                                <IPAddress>192.168.1.0</IPAddress>
                                <Wildcard>0.0.0.255</Wildcard>
                            </Network>
                        </Networks>
                    </Area>
                </Areas>
            </Ospf>
            <Rip>
                <Network>192.168.1.2</Network>
```

```
            </Rip>
        </Routing>
    </Router>
</Routers>
```

4.2 Rekonštrukcia topológie

Nato, aby sme mohli odsimulvať sieťovú topológiu, potrebujeme informácie ako daná topológia vyzerá. Vzhľadom k súčasnej implementácii simulátoru OMNeT++, je potreba informácie o topológii získať ešte pred spustením samotnej simulácie. Sú dva spôsoby, akými je možné preniesť informácie o topológii do simulácie.

- **IP adresy rozhraní** - na základe analýzy IP adries všetkých nastavených rozhraní na všetkých smerovačoch. Podľa IP adresy a masky, ktorá je nastavená na rozhraní sa hľadá rozhranie na inom smerovači, na ktorom je nastavená IP adresa s maskou tak, aby obe rozhrania patrili do rovnakej siete. Tento prístup predpokladá, že v sieti sa nebude vyskytovať smerovač, na ktorom je nastavená adresa rovnaká, ako na inom smerovači a taktiež, že neexistuje smerovač, ktorého rozhranie patrí do rovnakej siete ako rozhranie iného smerovača a zároveň medzi nimi neexistovalo spojenie. Tieto prípady by mohla spôsobiť prítomnosť NATu v sieti. NAT nie je v súčasnosti implementovaný v OMNeT++, preto zatiaľ túto možnosť neuvažujeme.

  Nevýhodou tohto prístupu je fakt, že pokiaľ užívateľ chce testovať nesprávne zapojenie topológie, nemá spôsob akým zadať topológiu, ktorú chce vyskúšať. Tento prístup spojí rozhrania, ktoré logicky patria k sebe.

- **externý súbor** - topológia je popísaná v externom súbore. Problém pri popise topológie, je akým spôsobom zadávajúci užívateľ identifikuje jedtnolivé smerovače, tak aby korenšpondovali s identifikáciou smerovačou v XML dokumente, ktorý uchováva nastavenia všetkých smerovačov. Tento spôsob nie je implementovaný v tejto práci.

V simulátore OMNeT++ sa topológia popisuje pomocou jazyka NED[4]. Tento popis je obsiahnutý v súbore .ned, ktorý je potrebný pre simuláciu. Skôr ako sme schopný vytvoriť informácie o tom ako topológia vyzerá, je nutné mať vytvorené nejaké dátové štruktúry, ktoré reprezentujú smerovač.

Smerovač je v simulátore OMNeT++ reprezentovaný ako modul. Celá sieť je popísaná jazykom NED a jej popis sa skladá z niekoľkých častí.

Popis začína kľúčovým slovom `network` za ktorým nasleduje pomenovanie siete. V sekcií `types` je možné si zadefinovať linky a ich parametre ako rýchlosť, alebo oneskorenie. Dôležitá sekcia v popise siete je `submodules`. V nej nastavujeme parametre jednotlivých modulov, z nášho pohľadu smerovačov. Simulátor OMNeT++ podporuje aj možnosť, že počet rozhraní jednotlivých smerovačov nie je dopredu známi. Ale v našom prípade je nutné pri parameteroch smerovača uviesť počet rozhraní, aby sme vedeli k rozhraniu priradiť nastavenia z XML dokumentu.

Ako posledná sekcia popisu siete je sekcia `connections`. V nej sa popisuje, ktoré rozhrania, ktorého smerovača sú pospájané a pomocou akej linky. Ako vyzerá popis jednoduchej siete si uvedieme na príklade.

```
network MyNetwork
{
```

```
    types:
        channel C extends DatarateChannel
        {
            datarate = 100Mbps;
        }
    submodules:
        R1: ANSARouter {
            parameters:
                @display('p=216,22');
            gates:
                ethg[2];
        }
        R2: ANSARouter {
            parameters:
                @display('p=152,92');
            gates:
                ethg[1];
                pppg[1];
        }
        R3: ANSARouter {
            parameters:
                @display('p=262,92');
            gates:
                ethg[1];
                pppg[1];
        }
    connections:       
        R1.ethg[0] <--> C <--> R2.ethg[0];
        R1.ethg[1] <--> C <--> R3.ethg[0];
        R2.pppg[0] <--> C <--> R3.pppg[0];
}
```

V príklade je vidieť, že v popise siete je možné nastaviť aj iné parametre, ako napr. súradnice umiestnenia smerovača na pracovnej doske.

V sekcií `connections` je dôležité si uvedomiť, že označenie rozhrania napr. `ethg[1]` je označenie pre rozhranie pomenované `eth1`. Preto bolo potrebné rozhrania z reálnych nastavení premenovať.

Pomenovania `R1, R2, R3` sú len v rámci simulácie, sú to identifikátory modulov, ktoré reprezentujú smerovače a nemajú nič spoločné s identifikáciou smerovačov v XML dokumente.

4.3 Nastavenia simulácie

Informácie o nastaveniach v XML dokumente a popis siete, nie sú dostatočné pre spustenie simulácie. Je potreba ešte vytvoriť základné informácie o tom z akého XML dokumentu sa majú načítať, ktorá sieť sa má odsimulovať.

Aby jednotlivé moduly, reprezentujúce smerovače vedeli, z ktorej časti XML načítať svoje nastavenia je potrebné každému smerovaču priradiť identifikátor.

Všetky tieto informácie sú uvedené v súbore *omnetpp.ini*, bez ktorého sa simulácia nespustí. Príklad ako vyzerá *omnetpp.ini*, ktorý je použitý na spustenie simulácie spoločne s .ned súborom uvedenom v podkapitole 4.2 vyzerá následovne:

```
[General]
description = 'Simple test'
network = SimpleTest
tkenv-plugin-path = ../../../etc/plugins

**.configFile ='xml.xml'
**.R1.routerId='10.1.13.1'
**.R2.routerId='172.16.0.1'
**.R3.routerId='172.15.0.1'
```

4.4 Zhrnutie

V tejto kapitole sme si uviedli, ktoré všetky informácie sú dôležité pre spustenie simulácie. Rozobrali nastavenia smerovačov, ktoré je potrebné nejakým spôsobom transformovať aby sme ich boli schopný odsimulovať. Nastavenia smerovačov sme uložili do univerzálnej XML štruktúry, ktorá je čitateľná pre OMNeT++.

Uviedli sme si, aké ostatné informácie okrem nastavenia smerovačou, sú potrebné pre spustenie simulácie a akým spôsobom ich získame.

Kapitola 5

Implementácia

V tejto časti popíšeme, akým spôsobom je implementovaný prekladač. Kapitolu rozdelíme na tri časti.

Prvá časť sa bude zaoberať implementáciou analyzátora, ktorý na základe gramatiky uvedenej v kapitole 3 bude rozpoznávať a kontrolovať syntax konfiguračného súboru Cisco smerovačov. Analyzátor budeme implementovať pomocou nástroja ANTLR (Another Tool for Language Recognition).

V druhej časti si uvedieme ako vyzerá práca s XML.

A v tretej časti popíšeme akým spôsobom spojíme analyzátor a XML.

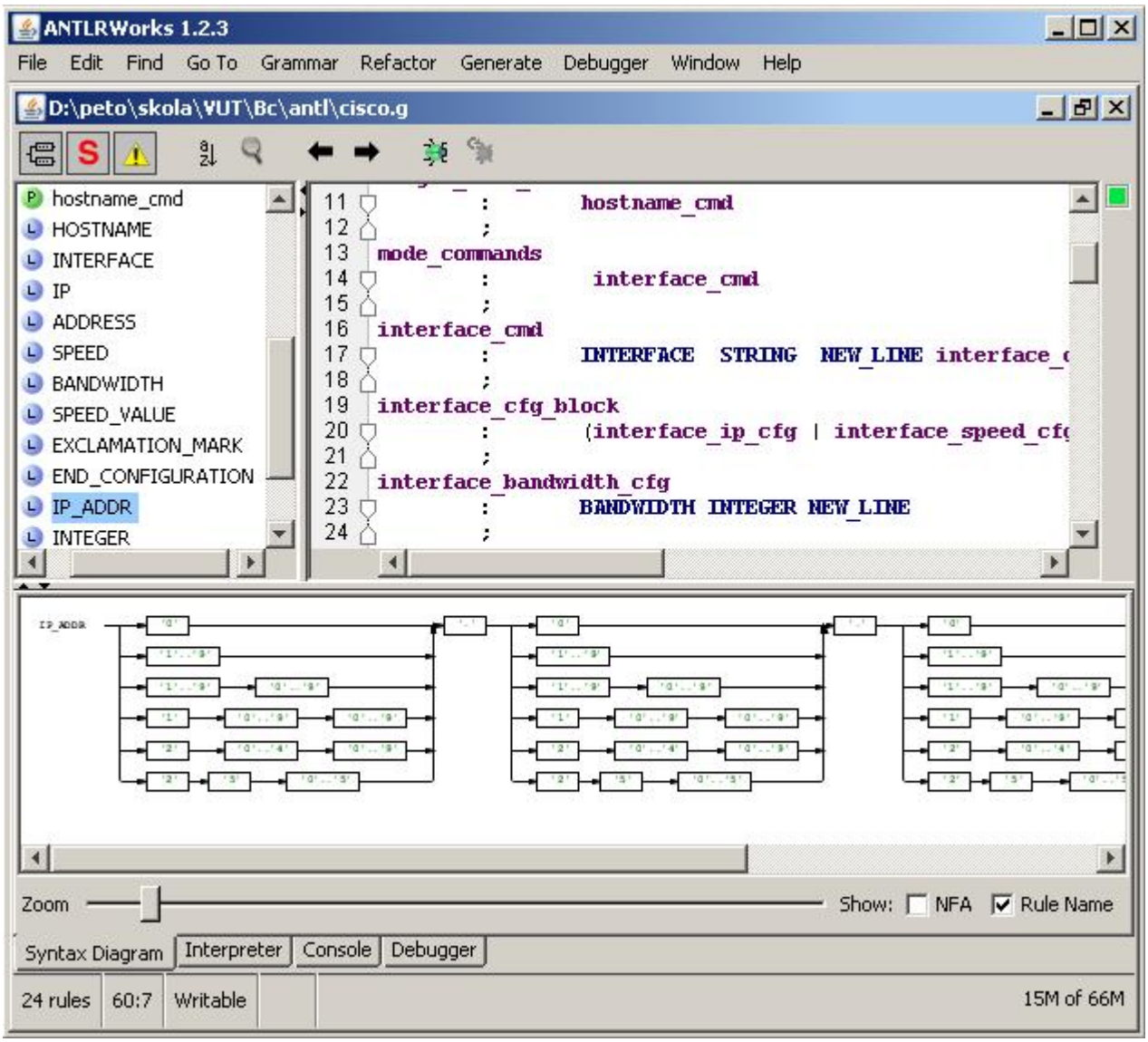
5.1 Analyzátor

Náš analyzátor sa skladá z dvoch častí, z lexikálneho a syntaktického analyzátora. Lexikálny analyzátor je prvá časť, ktorá zo zdrojového textu získava základne prvky (tokeny) jazyka a zasiela ich syntaktickému analyzátoru, ktorý na základe odvodzovacích pravidiel rozpozná či vstupný text odpovedá gramatike konfiguračného súboru.

5.1.1 Another Tool for Language Recognition

ANTLR je nástroj na prácu s jazykmi, ktorý poskytuje framework pre tvorbu prekladačov a kompilátorov na základe gramatického popisu. Na základe gramatiky ANTLR generuje analyzátor, ktorý rozpozná či vstupný text je syntakticky správne. Pridaním akcí k pravidlám popisujúcim jazyk sa z analyzátora ľahko môže stať interpret, alebo prekladač. ANTLR podporuje niekoľko cieľových jazykov, to znamená, že ANTLR vie vygenerovať analyzátor v rôznych jazykoch. Podporované sú *C, Java, C#, JavaScript, Delphi, Python.*

Pre prácu s ANTLR existuje vývojové prostredie ANTLRWorks s grafickým rozhraním. ANTLRWorks poskytuje kvalitný editor gramatík, ktorý priamo pri písaní gramatických pravidiel kreslí automat ktorý reprezentuje dané pravidlo. To pomáha pri vytváraní gramatiky a predchádza chybám. Taktiež pomáha predchádzať nedetermizmu, ktorý môže vzniknúť pri písaní gramatiky, zvýrazňovaním nedeterministických ciest. ANTLRWork obsahuje aj debugger, s ktorého pomocou je možné sledovať ako sú jednotlivé tokeny predávané syntaktickému analyzátoru a krok po kroku sledovať ako vytvára abstraktný syntaktický strom.



Obrázek 5.1: ANTLRWorks 1.2.3.

5.1.2 Práca s ANTLR

ANTLR pracuje pri popise gramatiky s EBNF, ktorá je mierne upravená. Zmena je len pri definícií a to namiesto = používa :. ANTLR používa syntax kde pravidlá pre lexikálny analyzátor začínajú vždy s veľkým písmenom a pravidlá pre syntaktický analyzátor začínajú vždy s malým písmenom.

Na začiatku každého popisu gramatiky uviesť jej názov. Tento názov musí byť zhodný s názvom súboru v ktorom je gramatika uložená s koncovkou *.g*.

```
/*Súbor : cisco.g*/
grammar cisco;
```

Súbor obsahujúci gramatiku sa môže skladať z niekoľkých častí. Na začiatku môžu byť uvedené nastavenia (`options`), ako napr. cieľový jazyk v ktorom má byť analyzátor vygenerovaný. Ako defaultný je jazyk *Java*.

Kôli lepšej prehľadnosti je vhodné v sekcií `tokens` si zadefinovať kľúčové slová, ktoré budeme používať v gramatike. Pre lepšiu prehľadnosť si uvedieme príklad, kde v nástroji ANTLR popíšeme základ gramatiky s jednoriadkovým pravidlom, ktorý sme si uviedli v podkapitole 3.2.3.

```
grammar cisco;
options{
  language = C;
  }
tokens{
  HOSTNAME = 'hostname';
}
configuration_file
        :  (comment | single_line_commands | mode_commands)*
                    END_CONFIGURATION  EOF
        ;
comment : EXCLAMATION_MARK STRING* NEW_LINE
        ;

single_line_commands
        : hostname_cmd {printf('Hello this is my grammar');}
        ;
mode_commands : ;

hostname_cmd
        : HOSTNAME STRING NEW_LINE
        ;
EXCLAMATION_MARK
        : '!'
        ;
END_CONFIGURATION
        : 'end' NEW_LINE
        ;
NEW_LINE: '\n' | '\r' | '\r\n'
        ;
```

```
WS      : '\t' | ' ' {channel=HIDDEN;}
        ;
STRING  : ANY_CHAR+
        ;
ANY_CHAR: ~(NEW_LINE | WS)*
        ;
```

V uvedenom príklade sú zaujímavosti, ktoré si treba všimnúť. Lexikálny a syntaktický analyzátor spolu komunikujú na rôznych kanáloch. Tokeny je možno zasielať aj na takzvaný schovaný kanál, to znamená že tokeny `WS` nebudú brané v úvahu v pravidlách syntaktického analyzátora.

Dôležitá vec je vkladanie kódu v cieľovom jazyku, ktorý je určený v sekcií `options`. Do sekcie označenej {} môžme vkladať ľubovolný kód. Pri pravidle `single_line_commands` sa do príkazovej riadky vypíše text *Hello this is my grammar*. Tieto sekcie kódu je možné vložiť kamkoľvek, ale záleží na pozícií kde je kód vložený (ako pozícia je myslené v akom poradí medzi tokenmi a pravidlami).

V momente keď je gramatika popísaná a odladená môžme vygenerovať kód lexikálneho a syntaktického analyzátoru. To dosiahneme pomocou klávesovej skratky `alt + shift + G`, alebo v menu cez položku *Generate*.

5.2 Reprezentácia dát

V podkapitole 5.1 sme si popísali akým spôsobom vytvoriť lexikálny a syntaktický analyzátor. Ten však zatiaľ len rozoznáva či vstupný text je platný v rámci gramatiky, na základe ktorej je vytvorený. Aby sme mohli na základe konfiguračného súboru vytvoriť XML štruktúru, je poterba vstupné dáta nejakým spôsobom transformovať.

Predtým ako vytvoríme samotné XML, dáta uložíme do objektu `xmlConfiguration`, ktorý zapúzdruje stromovú štruktúru a operácie nad ňou. Nad týmto stromom budeme prevádzať sémantickú kontrolu a transformácie nastavení, ktoré sú potrebné pre simulátor OMNeT++.

Tento objekt `xmlConfiguration` zapúzdruje aj výsledné vytvorenie XML dokumentu. Na prácu s XML používa C++ knižnicu TinyXML verzie 2.5.3, ktorá slúži na čítanie aj vytváranie XML.

Popíšeme si metódy, ktoré budeme používať pri vytváraní stromu.

- `goToElement(tXMLnode *ptr)` - metóda presunie kurzor na uzol stromu, na ktorý ukazuje ukazateľ prebraný ako parameter. Návratová hodnota je ukazateľ na aktuálnu pozíciu kurzoru v strome.
- `insertChild(string name,string id,string value,string txt,bool unique)` - metóda vytvorí potomka uzlu, na ktorý práve ukazuje kurzor. Návratová hodnota je ukazateľ na nový uzol.
  - `name` - určuje pomenovanie uzlu. Táto hodnota sa použije pri vytváraní XML na pomenovanie elementu.
  - `id` - nastavuje názov identifikátor uzlu. Pokiaľ je parameter zadaný ako prázdny reťazec, nenastaví sa žiadny identifikátor. Pri vytváraní XML sa na základe tohto identifikátoru vytvorí atribút, ktorý jednoznačne označí elementy s rovnakým rodičom (napríklad pomenovanie pri rozhraniach).

  - `value` - hodnota, ktorá bude obsiahnutá v identifikátore pomenovanom pomocou `id`.
  - `txt` - určuje textový reťazec ktorý bude uvedený vo výslednom XML dokumente. Pokiaľ element nemá obsahovať žiadny text, v `txt` bude predaný prázdny reťazec.
  - unique - hodnota určuje, či novo vytváraný uzol má mať pomenovanie uvedené v `name` jedinečné medzi uzlami s rovnakým rodičom. Ak sa vyskytuje medzi uzlami s rovnakým rodičom uzol s daným pomenovaním, metoda vráti ukazateľ na uzol, ktorý je už v strome obsiahnutý.

- `getRoot()` - metoda vráti ukazateľ na koreňový uzol stromu.
- `setAttribute(string name,string value)` - metóda nastaví atribút pre XML element.
- `saveXml(string file)` - zo stromu vytvorí XML štruktúru, ktorá reprezentuje nastavenia a uloží do súboru určenom parametrom `file`.
- `void transformForOmnet()` - metóda transformuje nastavenia, ktoré sú uvedené v časti 4.1.1 na nastavenia vhodné pre OMNeT++.

5.3 Spojenie XML a gramatiky

Na základe informácií v podkapitolách 5.1 a 5.2 sme schopný vytvoriť prekladač, ktorý skontroluje syntax konfiguračného súboru a pretransformuje nastavenia do XML štruktúry. Ako sme si uviedli v podkapitole 5.1 je možné k odvodzovacím pravidlám v gramatike dopísať akcie v cieľovom jazyku. Túto možnosť využijeme na dopísanie akcií pre vytvorenie XML.

V analyzátore je vytvorený ukazateľ `object`, ktorý vždy obsahuje inicializovaný objekt `xmlConfiguration`. V príklade z podkapitoly 5.1, sme pri pravidle `hostname_cmd` vytlačili text do konzoly. To teraz nahradíme za akcie, ktoré vytvoria časť XML štruktúry.

```
hostname_cmd
        : HOSTNAME name=STRING NEW_LINE
          {
            object->goToElement(object->getRoot());
            object->insertChild('Hostname','','',$name.text->chars,true)
          }
        ;
```

V nástroji ANTLR je možné označiť tokeny pomocou takzvaného *label*. Cez toto označenie je potom možné sa na token odkazovať cez tento *label*.

5.4 Prekladač

Pridaním akcií v podobe metód objektu, ktorý reprezentuje stromovú štruktúru, do gramatiky sme vytvorili prekladač, ktorý zo vstupného súboru vytvorí stromovú štruktúru.

Predtým ako sú tieto informácie uložené do XML dokumentu, je potrebné nastavenia pomocou metódy `transformForOmnet()` transformovať nastavenia pre OMNeT++. V poslednej fáze je vytvorený XML dokument pomocou metódy `saveXml(string file)`.

Prekladač môže pracovať v dvoch režimoch. Prvý režim kontroluje konfiguračný súbor detailne. To znamená, že prijíma len konfigurácie, ktoré implementuje simulátor OMNeT++. Ak sa vyskytne nastavenie, ktoré nepodporuje OMNeT++, prekladač skončí so syntaktickou chybou.

Druhý režim povoľuje aj iné príkazy, tie však nezahŕňa do konfigurácie.

Tieto dva režimy sú implementované pridaním akcií k pravidlu `unsupported`. V režime kedy akceptuje len implementované nastavenia pri vyvolaní tohto pravidla pri syntaktickej analýze prekladač skončí s ohlásením chyby.

Režim, v ktorom ma prekladač pracovať sa vyberá pomocou prepínača pri spúšťaní prekladača.

5.5 Zhrnutie

V tejto kapitole sme sa zoznámili s nástrojom ANTLR, za ktorého pomoci budeme na základe gramatiky generovať lexikálny a syntaktický analyzátor. Ukázali sme si akým spôsobom sa pracuje s objektom, ktorý reprezentuje stromovú štruktúru XML. A v poslednej časti sme spojením analyzátora a objektu `xmlConfiguration` vytvorili prekladač, ktorý vstupný konfiguračný súbor prevedie na XML dokument.

Kapitola 6

Rozšírenie o nové príkazy

Snahou tejto práce je vytvoriť prekladač, ktorý vytvorí nastavenia pre simuláciu z nastavení Cisco smerovačov a bol jednoducho rozšíriteľný o podporu nových príkazov.

V tejto kapitole si na konkrétnom príklade rozšírime prekladač o nové nastavenia, ktoré v súčastnosti nie sú podporované v OMNeT++.

Rozširovanie môžme rozdeliť na niekoľko častí. Tieto časti si postupne rozoberieme. Prekladač budeme rozširovať o základné nastavenia smerovacieho protokolu EIGRP.

6.1 Analýza príkazu

Ako prvý krok pri rozširovaní je analýza príkazu o ktorý chceme prekladač rozšíriť na základe Cisco dokumentácie [3]. Je potrebné si uvedomiť, ktoré nastavenia EIGRP sú podporované v simulátore OMNeT++. Pre tento príklad budeme uvažovať len nastavenie siete pomocou príkazu `network`. Príklad jednoduchého nastavenia je následovný.

```
router eigrp 1
 network 192.168.1.0 0.0.0.255
 network 192.168.2.0
!
```

6.2 Vytvorenie pravidiel

Ďalšia časť je vytvorenie gramatických pravidiel, ktoré rozpoznávajú nastavenia smerovacieho protokolu EIGRP. V tejto časti si tieto pravidla popíšeme pomocou EBNF spôsobom, aký sme si popísali v časti 3.2.3.

```
router    = ROUTER, WS, (
            'eigrp', WS, NUMBER, NE_WLINE, eigrp_cfg_block, NEW_LINE
          | (*Priestor pre ďalšie protokoly*)
          );
eigrp_cfg_block = {eigrp_network | unsupporter};
eigrp_network = 'network', WS, IP_ADDR, WS, IP_ADDR,
                [WS], NEW_LINE;

ROUTER = 'router';
```

6.3 Miesto pre uloženie

Aby sme mohli previesť nastavenia do simulácie, je potrebné zadefinovať, ako bude vyzerať XML štruktúra pre protokol EIGRP. Tieto nastavenia zaradíme do sekcie nastavení smerovania.

Príklad uloženia nastavení v XML.

```
<Routers>
    <Router id='0.0.0.0'>
        <Routing>
            <EIGRP>
              <AS id='1'>
                <Network addr='192.168.1.0' wc='0.0.0.255' />
                <Network addr='192.168.2.0' />
              </AS>
            </EIGRP>
        </Routing>
    </Router>
</Routers>
```

6.4 Rozšírenie implementácie

Posledná časť je pomocou vývojového prostredia ANTLRWorks rozšíriť súčasny prekladač. Vychádzame zo súčasnej implementácie, ktorá už obsahuje kľúčové slová `router` a `network`.

```
// Do sekcie tokens pridáme kľúčové slovo eigrp
tokens {
  EIGRP = 'eigrp';
}
// Pravidlo router rozšírime o pravidlo pre EGIRP
router : ROUTER
            { // Najdeme v strome uzol 'Routing'
                 object->goToElement(object->getRoot());
          object->goToElement(object->insertChild('Routing','','','',true));
            }
    (
            OSPF DIGITS NEWLINE ospf_cfg_block
        |    RIP NEWLINE rip_cfg_block
        |    EIGRP as=DIGITS
      {
        // Vytvoríme uzol 'EIGRP' a pod nim 'AS'
        object->goToElement(object->insertChild('EIGRP','','','',true));
        object->goToElement(object->insertChild('AS','id',
                            to_string(as.text->chars),'',false));
      }
            NEWLINE eigrp_cfg_block
    )
            ENDCOMMAND
       ;
```

```
eigrp_cfg_block
  :    (eigrp_newtork | unsupported)*
  ;
eigrp_newtork
  :    NETWORK ip=IP_ADDR wc=IP_ADDR NEWLINE
    {
      // Premenna tmp je pomocný ukazatel, ktorý je už definovaný
      tmp = object->current;
      object->goToElement(object->insertChild('Network','','','',false));
      object->setAttribute('addr',to_string(ip.text->chars));
      object->setAttribute('wc',to_string(wc.text->chars));
      object->current = tmp;
    }
    ;
```

V prostredí ANTLRWorks pomocou klávesovej skratky `alt + shift + G` vygenerujeme zdrojový kód pre lexikálny a syntaktický analyzátor, ktorý pomocou *Makefile* môžme preložiť a výsledkom je nový prekladač rozšírený o nastavenia EIGRP.

Kapitola 7

Záver

Cieľom bakalárskej práce bolo vytvoriť nástroj, ktorý by umožňoval automatický preklad konfiguračných súborov Cisco, na základe ktorého by bolo možné vytvoriť model siete v simulátore OMNeT++. Tento nástroj bol vytváraný ako prekladač, ktorý na základe gramatiky vytvorenej zo syntaxe príkazov, prevedie nastavenia do XML formátu a vytvorí nastavenia potrebné pre spustenie simulácie. Pri vytváraní gramatiky sa uvažovali nastavenia pre rozhrania, hostname, smerovanie (OSPF, RIP, statické smerovanie) a ACL.

7.1 Ďalší vývoj

V súčastnosti je implementovaná rekonštrukcia na základe IP adries. Ďalší vývoj by sa mohol zaoberať externým popisom sieťovej topológie. Popis topológie by bol uvedený v externom súbore a tým by sa umožnilo simulovať aj chybné zapojenie siete.

Literatura

[1] Parr, T.: *The Definitive ANTLR Reference*. The Pragmatic Bookshelf, 2008, iSBN-10: 0-9787392-5-6.

[2] WWW stránky: Cisco Command References, Part 1. http://www.cisco.com/en/US/docs/ios/12_0/np1/command/reference/np1_r.html.

[3] WWW stránky: Cisco IOS IP Command Reference. http://www.cisco.com/en/US/docs/ios/12_3/ipaddr/command/reference/ipras_r.html.

[4] WWW stránky: NED Reference. http://omnetpp.org/doc/omnetpp40/manual/usman.html#sec426, [cit. 17. 05. 2009].

[5] WWW stránky: OMNeT++ User Manual. http://omnetpp.org/doc/omnetpp40/manual/usman.html.

[6] WWW stránky: Syntactic metalanguage - Extended BNF. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip.

[7] WWW stránky: TinyXML Documentation. http://www.grinninglizard.com/tinyxmldocs/index.html.

Dodatek A

Obsah CD

Obsah priloženého CD:

- **zdrojové súbory textu práce** - v adresári *sprava*
- **zdrojové súbory OMNeT++** - v adresári *omnet*
- **zdrojové súbory prekladača** - v adresári *src*
- **ukážky prekladov** - v adresári *example*
- **preložený prekladač pre Windows** - v adresári *bin*
- **ANTLR a ANTLRWorks** - v adresári *antlr*

Dodatek B

Manual

`prekladac.exe` je konzolová aplikácia. Použitie `prekladac.exe [-pedantic=true|false][-o][-t] files`

- `-pedantic` - parameter určuje či nepodporované nastavenia ignorovať, alebo kontrolovať.
- `-t` - parameter určuje či sa má rekonštruovať topológia na základe IP adries.
- `-o` - definuje výstupný súbor pre XML

Dodatek C

Prehľad podporovaných príkazov

Podporované príkazy		
Príkaz	Parametrujúci príkaz	Podpora syntaxe
hostname	-	úplná
interface	ip address	úplná
	duplex	úplná
	bandwidth	úplná
	speed	úplná
	ip ospf ospf-cost	úplná
	ip ospf priority	úplná
	ip ospf hello-interval	úplná
	ip ospf dead-interval	úplná
	ip ospf hello-interval	úplná
	ip ospf authentication	úplná
	shutdown	úplná
	ip access-group	čiastočná
ip route	-	čiastočná
access-list	-	čiastočná
ip access-list	-	čiastočná
router rip	network	úplná
	passive-interface	úplná
	redistribude	čiastočná
router ospf	network	úplná
	area	čiastočná

Dodatek D

Gramatika IOS

```
configuration_file = (comment | single_line_commands | mode_commands)*,
                     END_CONFIGURATION, EOF;
comment = EXCLAMATION_MARK, {WS | STRING}, NEW_LINE;

single_commands = hostname_cmd | acl_simple;
mode_commands = interface_cmd | acl_block;

interface_cmd = 'interface', WS, STRING, [WS], NEW_LINE,
                interface_cfg_block, WS, EXCLAMATION_MARK, [WS], NEW_LINE;
interface_cfg_block = {(interface_ip_cfg | interface_speed_cfg |
                      interface_bandwidth_cfg | interface_duplex_cfg |
                      if_ospf_cost | if_ospf_priority | if_ospf_hellointerval |
                      if_ospf_deadinterval |if_ospf_authentication |
                      if_ospf_authentication | if_shutdown | if_acl_group )};
interface_ip_cfg = 'ip', WS, 'address', WS, IP_ADDR, WS, IP_ADDR, [WS], NEW_LINE;
interface_speed_cfg = 'speed', WS, SPEED_VALUE, [WS], NEW_LINE;
interface_bandwidth_cfg =   'bandwidth', WS, NUMBER, [WS], NEW_LINE;
interface_duplex_cfg =   'duplex', WS, ('full' | 'half' | 'auto'),[WS], NEW_LINE;
if_ospf_cost = 'ip', WS, 'ospf', WS, 'cost', WS, NUMBER, [WS], NEW_LINE;
if_ospf_priority  =  'ip', WS, 'ospf', WS, 'priority', WS, [WS], NUMBER,
                     [WS], NEW_LINE;
if_ospf_hellointerval = 'ip', WS, 'ospf', WS, 'hello-interval', WS,
                        [WS], NUMBER, [WS], NEW_LINE;
if_ospf_deadinterval = 'ip', WS, 'ospf', WS, 'hello-dead', WS,
                        [WS], NUMBER, [WS], NEW_LINE;
if_ospf_authentication = 'ip', WS, 'ospf', WS, 'authentication', WS,
                        [WS], ('null'|'clear-text'|'message-digest'),
                        [WS], NEW_LINE;
if_shutdown = 'shutdown' | ('no', WS, 'SHUTDOWN'), [WS], NEW_LINE;
if_acl_group = 'ip', WS, 'access-group', WS, STRING | NUMBER, WS, 'in'|'out',
               [WS], NEW_LINE;
hostname_cmd = 'hostname', WS, STRING, [WS], NEW_LINE;
acl_simle = 'access-list', WS, NUMBER, (acl_standard|acl_extended), [WS],
            NEW_LINE;
acl_standard = 'permit'|'deny', WS, acl_addr_spec, [WS], NEW_LINE;
```

```
acl_extended = 'permit'|'deny' (acl_extended_ip | acl_extended_icmp |
                acl_extended_tcp | acl_extended_udp);
acl_extended_ip = 'ip', WS, acl_addr_spec, WS, acl_addr_spec, [WS];
acl_extended_icmp = 'icmp', WS, acl_addr_spec, WS, acl_addr_spec, [WS];
acl_extended_tcp = 'tcp', WS, acl_addr_spec, [acl_operator], WS,
                   [acl_operator], acl_addr_spec, [WS];
acl_extended_udp = 'tcp', WS, acl_addr_spec, [acl_operator], WS,
                   [acl_operator], acl_addr_spec, [WS];
acl_block = 'ip', WS, 'access-list', WS, ('standard', WS, STRING, [WS],
            NEW_LINE, acl_standard, {acl_standard}) | ('extended',
            WS, STRING, [WS], NEW_LINE, acl_extended, {acl_extended});
acl_operator = ('eq'|'lt'|'gt'|'neq'), WS, ('ftp'|'www'|'smtp'|NUMBER);
acl_addr_spec = 'any' | (IP_ADDR, WS, IP_ADDR) | ('host', WS, IP_ADDR);

router = 'router', WS, ('ospf', WS, NUMBER, [WS], NEW_LINE, ospf_cfg_block)
         | ('rip', WS, NEW_LINE, rip_cfg_block)
ospf_cfg_block = {ospf_network};
ospf_network = 'network', WS, IP_ADDR, WS, IP_ADDR, WS, 'area', WS,
               NUMBER | IP_ADDR, NEW_LINE;
rip_cfg_block = {rip_network | rip_passive_int | rip_redistribute};
rip_network = 'network', WS, IP_ADDR, [WS], NEW_LINE;
rip_passice_int = 'passive-interface',WS ,STRING ,[WS], NEW_LINE;
rip_redistribude = 'redistribude', WS, 'ospf', NUMBER, [WS], NEWLINE;
SPEED_VALUE = 'auto' | '10' | '100';
EXCLAMATION_MARK = '!';
END_CONFIGURATION = 'end', [WS], NEW_LINE;
NUMBER = DIGIT,{DIGIT};
DIGIT = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
NEW_LINE = '\n' | '\r' | '\r\n';
WS = ('\t' | ' '), {('\t' | ' ')};
IP_ADDR = 3* (('0' | ('1'..'9') | ('1'..'9')('0'..'9') | '1'('0'..'9')('0'..'9') |
          '2'('0'..'4')('0'..'9') | '2''5'('0'..'5'))'.')
          ('0' | ('1'..'9') | ('1'..'9')('0'..'9') | '1'('0'..'9')('0'..'9') |
          '2'('0'..'4')('0'..'9') | '2''5'('0'..'5'));
STRING = ANY_CHAR, {ANY_CHAR};
ANY_CHAR = -(NEW_LINE | WS)*;
```

Dodatek E

Príklad prekladu a jeho výstup

R1 :

```
!
hostname R1
!
interface FastEthernet0/0
 ip address 10.1.12.1 255.255.255.0
 duplex auto
 speed auto
!
interface Serial0/1
 ip address 10.1.13.1 255.255.255.0
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 10.0.0.0 0.255.255.255 area 0
!
router rip
 network 10.0.0.0
!
end
```

R2 :

```
!
hostname R2
!
interface FastEthernet0/0
 ip address 10.1.12.2 255.255.255.0
 duplex auto
 speed auto
!
interface Serial0/1
 ip address 10.1.13.2 255.255.255.0
```

```
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 10.0.0.0 0.255.255.255 area 0
!
router rip
 network 10.0.0.0
!
end
```

Prekladač spustíme s parametrami :

```
-pedantic=false -t -o test.xml R1.txt R2.txt
```

Obsah test.xml :

```
<?xml version='1.0' ?>
<Routers>
    <Router id='10.1.13.1'>
        <Hostname>R1</Hostname>
        <Interfaces>
            <Interface name='eth0'>
                <IPAddress>10.1.12.1</IPAddress>
                <Mask>255.255.255.0</Mask>
                <Duplex>auto</Duplex>
                <Speed>auto</Speed>
            </Interface>
            <Interface name='ppp0'>
                <IPAddress>10.1.13.1</IPAddress>
                <Mask>255.255.255.0</Mask>
                <Duplex>auto</Duplex>
                <Speed>auto</Speed>
            </Interface>
        </Interfaces>
        <Routing>
            <Ospf>
                <RFC1583Compatible />
                <Areas>
                    <Area id='0.0.0.0'>
                        <Networks>
                            <Network>
                                <IPAddress>10.0.0.0</IPAddress>
                                <Wildcard>0.255.255.255</Wildcard>
                            </Network>
                        </Networks>
                    </Area>
                </Areas>
            </Ospf>
```

```
        </Routing>
        <Routing>
            <Rip>
                <Network>10.0.0.0</Network>
            </Rip>
        </Routing>
    </Router>
    <Router id='10.1.13.1'>
        <Hostname>R1</Hostname>
        <Interfaces>
            <Interface name='eth0'>
                <IPAddress>10.1.12.1</IPAddress>
                <Mask>255.255.255.0</Mask>
                <Duplex>auto</Duplex>
                <Speed>auto</Speed>
            </Interface>
            <Interface name='ppp0'>
                <IPAddress>10.1.13.1</IPAddress>
                <Mask>255.255.255.0</Mask>
                <Duplex>auto</Duplex>
                <Speed>auto</Speed>
            </Interface>
        </Interfaces>
        <Routing>
            <Ospf>
                <RFC1583Compatible />
                <Areas>
                    <Area id='0.0.0.0'>
                        <Networks>
                            <Network>
                                <IPAddress>10.0.0.0</IPAddress>
                                <Wildcard>0.255.255.255</Wildcard>
                            </Network>
                        </Networks>
                    </Area>
                </Areas>
            </Ospf>
        </Routing>
        <Routing>
            <Rip>
                <Network>10.0.0.0</Network>
            </Rip>
        </Routing>
    </Router>
</Routers>
```

Obsah omnetpp.ini :

```
[General]
```

```
description = 'MyNetwork'
network = MyNetwork
tkenv-plugin-path = ../../../etc/plugins

**.configFile ='test.xml'
**.R1.routerId='10.1.13.1'
**.R2.routerId='10.1.13.1'
**.localPort = 520
**.destPort = 520
```

Obsah MyNetwork.ned :

```
package /*vlozit cestu s umiestnenim simulacie*/ ;
import inet.ansa.ANSARouter;
import inet.world.ChannelInstaller;
import ned.DatarateChannel;
import inet.nodes.inet.StandardHost;

network MyNetwork
{
types:
channel C extends DatarateChannel
{
delay = 0.1us;
}
submodules:
R1 : ANSARouter {
gates:

ethg[1];
pppg[1];
}
R2 : ANSARouter {
gates:

ethg[1];
pppg[1];
}
connections:
R1.ethg[0] <--> C <--> R2.ethg[0];
R1.pppg[0] <--> C <--> R2.pppg[0];
}
```